

# 组合与继承

## (OOP)

黄民烈

[aihuang@tsinghua.edu.cn](mailto:aihuang@tsinghua.edu.cn)

<http://coai.cs.tsinghua.edu.cn/hm1>

课程团队：刘知远 姚海龙 黄民烈

# 上期要点回顾

- 拷贝构造函数：对象之间的拷贝
- 右值引用：延长临时对象的生命周期
- 移动构造函数：避免频繁的拷贝

# 本讲内容提要

- 组合
- 继承
- 成员访问权限
- 重写隐藏与重载
- 多重继承

# 对象(类)之间的关系?

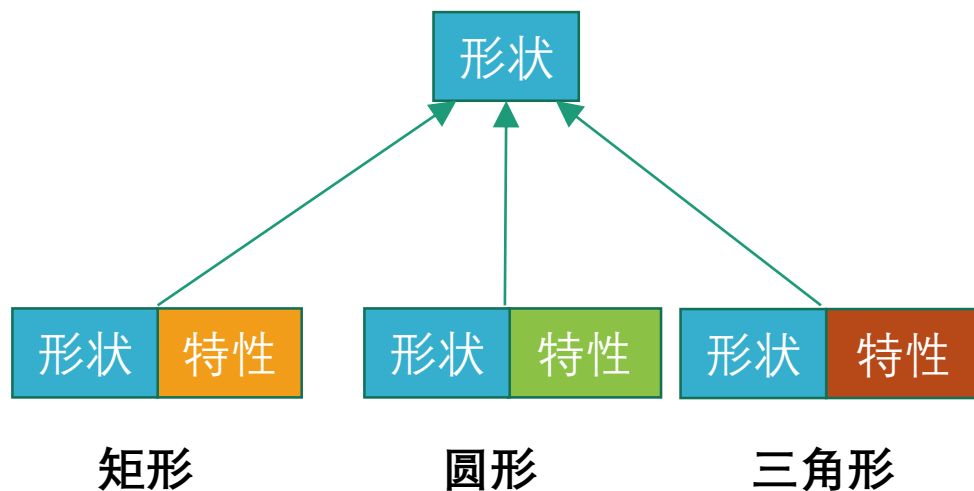
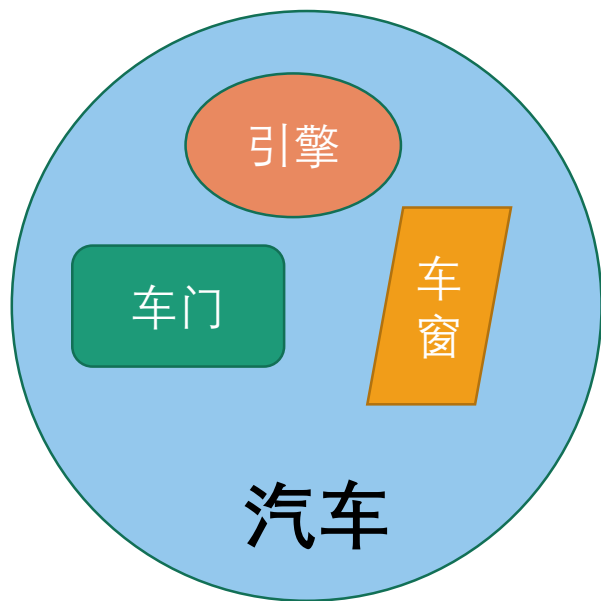
- 思考：这些是什么关系？
- 汽车：车门、车窗、引擎、轮胎
- 形状：矩形，圆形，三角形，正方形

# 对象(类)之间的关系?

## ■ 思考：这些是什么关系？

- **has-a**: 车门，车窗，引擎是汽车的**组成部分**
- **is-a**: 矩形，圆形，三角形是一种**特殊**的形状

## ■ 区分：“整体-部分” vs. “一般-特殊”



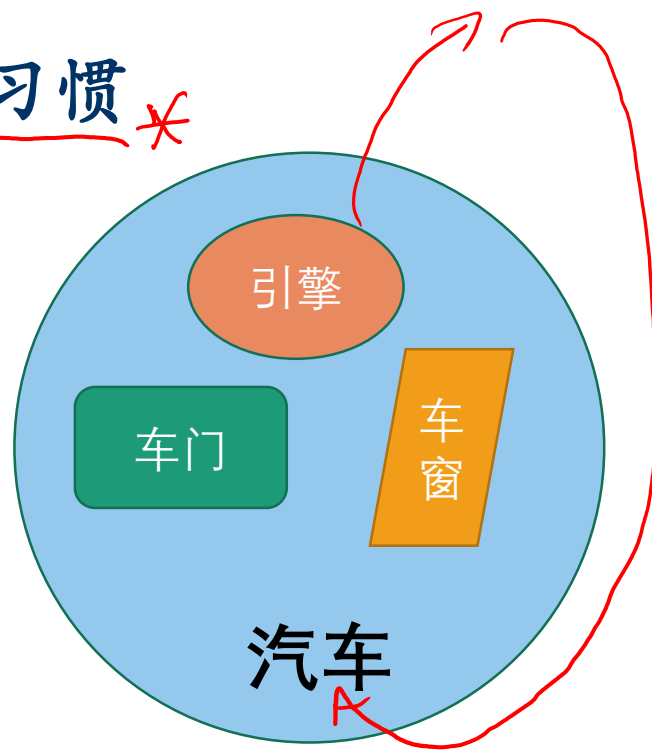
# 组合

■ has-a: 如果对象a是对象b的一个组成部分，则称b为a的整体对象，a为b的部分对象。并把b和a之间的关系，称为“**整体一部分**”关系（也可称为“**组合**”或“**has-a**”关系）。

■ 程序设计反映对客观世界的认知习惯 \*

■ 对象组合的两种实现方法：

- 已有类的对象作为新类的**公有**数据成员，这样通过允许直接访问子对象而“提供”旧类接口
- 已有类的对象作为新类的**私有**数据成员。新类可以调整旧类的**对外接口**，可以不使用旧类原有的接口（相当于对接口作了转换）



# 对象组合示例

```
#include <iostream>
using namespace std;

class Wheel{
    int _num; 私有
public:
    void set(int n){_num=n;}
};

class Engine{
public:
    int _num; 私有
    void set(int n){_num=n;}
};
```

# 对象组合示例

```
class Car{  
private:  
  Wheel w;  
public:  
  Engine e; /// 公有成员，直接访问其接口  
  void setWheel(int n){w.set(n);} /// 提供私有成员的访问接口  
};
```

```
int main()  
{
```

```
  Car c;
```

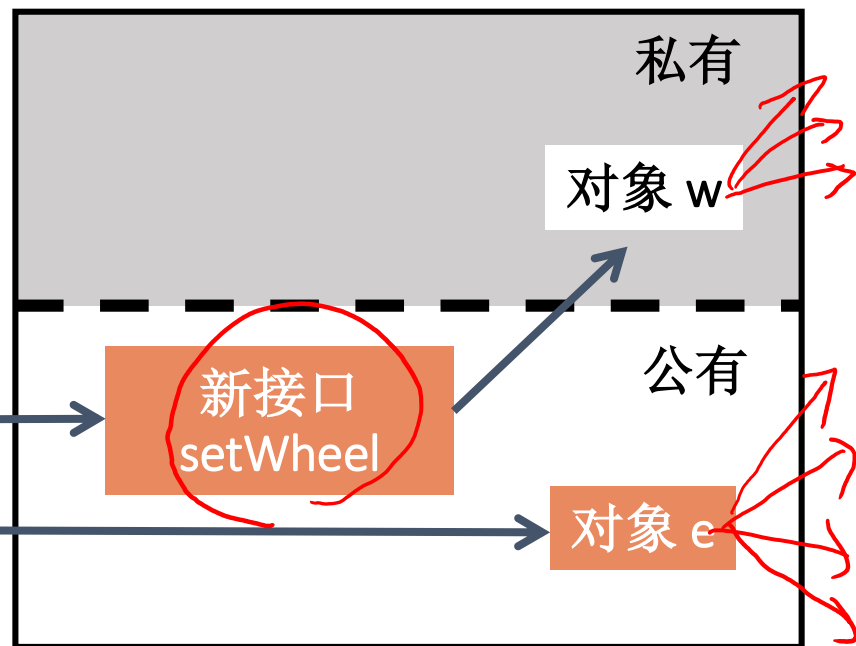
```
  c.e.set(1); 方法二：私有成员
```

```
  c.setWheel(4);
```

```
  return 0;
```

```
}
```

方法一：公有成员



新对象c（组合）



# 组合

Car

- 子对象构造时若需要参数，则应在当前类的构造函数的初始化列表中进行。若使用默认构造函数来构造子对象，则不用做任何处理。

- 课后尝试：修改代码，使得Wheel、Engine的构造函数带参数

## ■ 对象构造与析构函数的次序

- 先完成子对象构造，再完成当前对象构造
- 子对象构造的次序仅由在类中声明的次序所决定
- 析构函数的次序与构造函数相反

Car     car;

# 对象组合示例 构造与析构

```
#include <iostream>
using namespace std;
```

```
class S1 { //Single1类别
    int ID;
public:
    S1(int id) : ID(id) { cout << "S1(int)" <<
endl; }
    ~S1() { cout << "~S1()" << endl; }
};
```

```
class S2 {//Single2类别
public:
    ✓ S2() { cout << "S2()" << endl; }
    ✓ ~S2() { cout << "~S2()" << endl; }
};
```

# 对象组合示例 构造与析构

```
class C3 {//Composite3类别
```

```
    int num;
```

```
    S1 sub_obj1; /// 构造函数带参数
```

```
    S2 sub_obj2; /// 构造函数不带参数
```

```
public:
```

```
✓ C3() : num(0), sub_obj1(123) /// 构造函数初始化列表中构造子对象
```

```
    { cout << "C3()" << endl; }
```

```
✓ C3(int n) : num(n), sub_obj1(123)
```

```
    { cout << "C3(int)" << endl; }
```

```
✓ C3(int n, int k) : num(n), sub_obj1(k)
```

```
    { cout << "C3(int, int)" << endl; }
```

```
    ~C3() { cout << "~C3()" << endl; }
```

```
};
```

```
int main()
```

```
{
```

```
    C3 a, b(1), c(2), d(3, 4);
```

```
    return 0;
```

```
}
```

# 对象组合运行结果

**a** {  
 S1(int) ✓  
 S2() ✓  
 C3() ✗  
**b** {  
 S1(int) ✓  
 S2() ✓  
 C3(int) ✗  
**c** {  
 S1(int) ✓  
 S2() ✓  
 C3(int) ✗  
**d** {  
 S1(int) ✓  
 S2() ✓  
 C3(int, int) ✓  
 ~C3() ✗  
 ~S2()  
 ~S1()  
**d** {  
 ~C3()  
 ~S2()  
 ~S1()  
**b** {  
 ~C3()  
 ~S2()  
 ~S1()  
**a** {  
 ~C3()  
 ~S2()  
 ~S1()

```

class C3 {
    int num;
    S1 sub_obj1;
    S2 sub_obj2;
    .....
};
  
```

```

int main()
{
    C3 a, b(1), c(2), d(3, 4);
    return 0;
}
  
```

# 组合

## ■ 对象拷贝与赋值运算

- 如果调用拷贝构造函数且没有给类显式定义拷贝构造函数，编译器将自动合成：（1）对**有显式**定义拷贝构造函数的子对象调用该拷贝构造函数，（2）对**无显式**定义拷贝构造函数的子对象采用位拷贝。
- 赋值的默认操作类似

# 对象组合示例

## 默认拷贝与赋值

```
#include <iostream>
using namespace std;
```

```
class C1{
public:
    int i;
    C1(int n):i(n){}
    ✱ C1(const C1 &other) /// 显式定义拷贝构造函数
        {i=other.i; cout << "C1(const C1 &other)" << endl;}
};
```

```
class C2{
public:
    int j;
    C2(int n):j(n){}
    C2& operator= (const C2& right){/// 显式定义赋值运算符
        if(this != &right){
            j = right.j;
            cout << "operator=(const C2&)" << endl;
        }
        return *this;
    }
};
```

# 对象组合示例 默认拷贝与赋值

```
class C3{ // 复合类
public:
```

```
    ✓ C1 c1;
```

```
    ✓ C2 c2;
```

```
    C3():c1(0), c2(0){}
```

```
    C3(int i, int j):c1(i), c2(j){}
```

```
    void print(){cout << "c1.i = " << c1.i << " c2.j = " << c2.j << endl;}
```

```
};
```

```
int main(){
```

```
    C3 a(1, 2);
```

```
    C3 b(a); //C1执行显式定义的拷贝构造,
```

△ C2执行自动合成的拷贝构造

```
    cout << "b: ";
```

```
    b.print();
```

```
    cout << endl;
```

```
    C3 c;
```

```
    cout << "c: ";
```

```
    c.print();
```

```
    c = a; //C1执行自动合成的拷贝赋值,
```

C2执行显式定义的拷贝赋值

```
    cout << "c: ";
```

```
    c.print();
```

```
    return 0;
```

```
}
```

运行结果

C1(const C1 &other)  
b: c1.i = 1 c2.j = 2

c: c1.i = 0 c2.j = 0  
operator=(const C2&)  
c: c1.i = 1 c2.j = 2

位拷贝

关于下列代码的说法，错误的是（\n为换行符）

```
#include <iostream>
using namespace std;
class A {
    int data;
public:
```

```
✗ A():data(0)
    {cout << "A::A(" << data << ")\n";}
    A(int i):data(i)
    {cout << "A::A(" << i << ")\n";}
};
class B {
    int data{2018};
    A a;
```

```
public:
    ✓ B(){}
    ✓ B(int i):a(i){}
    void print()
    {cout << "data = " << data << endl;}
};
int main() {
    ✗ B obj1;
    B obj2(2019);
    obj1.print();
    obj2.print();
    return 0;
}
```

- ☐ A B类的默认构造函数没有显式调用A类的构造函数，此时编译器会自动调用A类的默认构造函数
- ☐ B B类的普通构造函数可以在初始化列表中显式调用A类的普通构造函数
- ☒ C 该程序的输出为 A::A(0)\nA::A(2019)\ndata = 2018\ndata = 2019\n 2018
- ☐ D obj1析构时先执行B类的析构函数，再执行A类的析构函数

提交



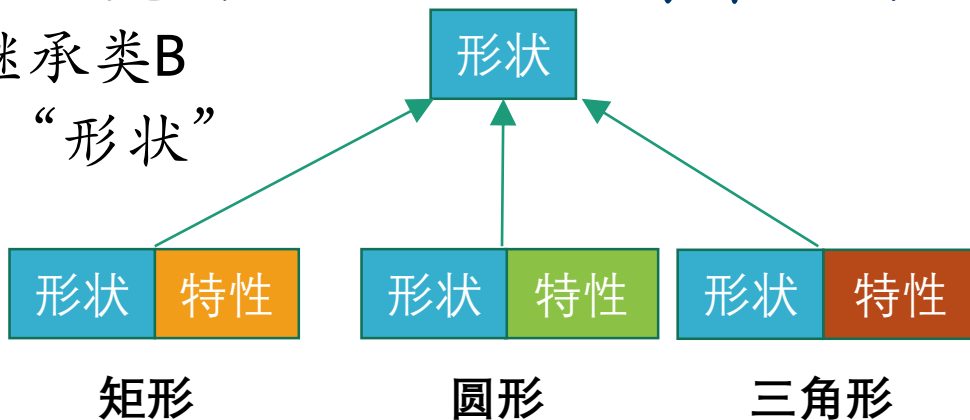
# 继承

■ **is-a**: “**一般—特殊**”结构，也称“**分类结构**”，是由一组具有“**一般—特殊**”关系的类所组成的结构。

- 如果类A具有类B全部的属性和服务，而且具有自己特有的某些属性或服务，则称A为B的特殊类，B为A的一般类。
- 如果类A的全部对象都是类B的对象，而且类B中存在不属于类A的对象，则A是B的特殊类，B是A的一般类。

■ **C++使用继承来表达类间的“一般—特殊结构”**

- 上述例子中类A继承类B
- “矩形” 继承 “形状”



# 继承

- 被继承的已有类，被称为**基类**(base class)，也称“父类”。
- 通过继承得到的新类，被为**派生类**(derived class，也称“**子类**”、“**扩展类**”。
- 常见的继承方式：public, private
  - `class Derived : [private] Base { .. };` 缺省继承方式为private继承。
  - `class Derived : public Base { ... };`
- **protected** 继承很少被使用
  - `class Derived : protected Base { ... };`

# 继承

## ■ 什么不能被继承？

- ✗ • **构造函数**：创建派生类对象时，必须调用派生类的构造函数，派生类构造函数调用基类的构造函数，以创建派生对象的基类部分。C++11新增了继承构造函数的机制（使用using），但默认不继承
- ✗ • **析构函数**：释放对象时，先调用派生类析构函数，再调用基类析构函数
- ✗ • **赋值运算符**：因为赋值运算符包含一个类型为其所属类的形参
- ✗ • **友元函数**：不是类成员

# 继承示例

```
#include <iostream>
using namespace std;

class Base{
public:
    int k = 0;
    void f(){cout << "Base::f()" << endl;}
    Base & operator= (const Base &right){
        if(this != &right){
            k = right.k;
            cout << "operator= (const Base &right)" << endl;
        }
        return *this;
    }
};

class Derive: public Base{};

int main(){
    Derive d;
    cout << d.k << endl; //Base数据成员被继承
    d.f(); //Base::f()被继承

    Base e;
    //d = e; //编译错误, Base的赋值运算符不被继承
    return 0;
}
```

## 运行结果

0  
Base::f()

# 派生类对象的构造与析构过程

- 基类中的数据成员，通过继承成为派生类对象的一部分，需要在构造派生类对象的过程中调用基类构造函数来正确初始化。
  - 若没有显式调用，则编译器会自动生成一个对基类的默认构造函数的调用。
  - 若想要显式调用，则只能在派生类构造函数的初始化成员列表中进行，既可以调用基类中不带参数的默认构造函数，也可以调用合适的带参数的其他构造函数。
- 先执行基类的构造函数来初始化继承来的数据，再执行派生类的构造函数。
- 对象析构时，先执行派生类析构函数，再执行由编译器自动调用的基类的析构函数。

# 调用基类构造函数

- 若没有显式调用，则编译器会自动生成一个对基类的默认构造函数的调用。

```
class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; } /// 默认构造函数
    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};
class Derive : public Base {
public:
    Derive() { cout << "Derive::Derive()" << endl; }
    /// 无显式调用基类构造函数，则调用基类默认构造函数
};
int main() {
    Derive obj;
    return 0;
} // g++ 1.cpp -o 1.out -std=c++11
```

**运行结果**

Base::Base(0)  
Derive::Derive()

# 调用基类构造函数

- 若想要显式调用，则只能在派生类构造函数的**初始化成员列表**中进行。

```
class Base
{
    int data;
public:
    Base() : data(0) { cout << "Base::Base(" << data << ")\n"; }
    // 默认构造函数

    Base(int i) : data(i) { cout << "Base::Base(" << data << ")\n"; }
};

class Derive : public Base {
public:
    Derive(int i) : Base(i) { cout << "Derjve::Derive()" << endl; }
    // 显式调用基类构造函数
};

int main() {
    Derive obj(356);
    return 0;
} // g++ 1.cpp -o 1.out -std=c++11
```

运行结果

Base::Base(356)  
Derive::Derive()

# 继承基类构造函数 (1)

- 在派生类中使用 using Base::Base; 来继承基类构造函数，相当于给派生类“定义”了相应参数的构造函数，如

```
class Base
{
    int data;
public:
    Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
};
class Derive : public Base {
public:
    using Base::Base; // 相当于 Derive(int i):Base(i){};
};
int main() {
    Derive obj(356);

    return 0;
} // g++ 1.cpp -o 1.out -std=c++11
```

运行结果

Base::Base(356)



# 继承基类构造函数 (2)

- 当基类存在多个构造函数时，使用using会给派生类自动构造多个相应的构造函数。

```
class Base
{
    int data;
public:
    △ Base(int i) : data(i) { cout << "Base::Base(" << i << ")\n"; }
    △ Base(int i, int j)
        { cout << "Base::Base(" << i << "," << j << ")\n"; }
};
class Derive : public Base {
public:
    using Base::Base;
};
int main() {
    △ Derive obj(356);
    △ Derive obj(356, 789);
    return 0;
} // g++ 1.cpp -o 1.out -std=c++11
```

///相当于 Derive(int i):Base(i){};  
///加上 Derive(int i, int j):Base(i, j){};


## 运行结果

Base::Base(356)  
Base::Base(356,789)

# 继承基类构造函数 (3)

- 如果基类的某个构造函数被声明为私有成员函数，则不能在派生类中声明继承该构造函数。
- 如果派生类使用了继承构造函数，编译器就不会再为派生类生成默认构造函数。

以下说法正确的是

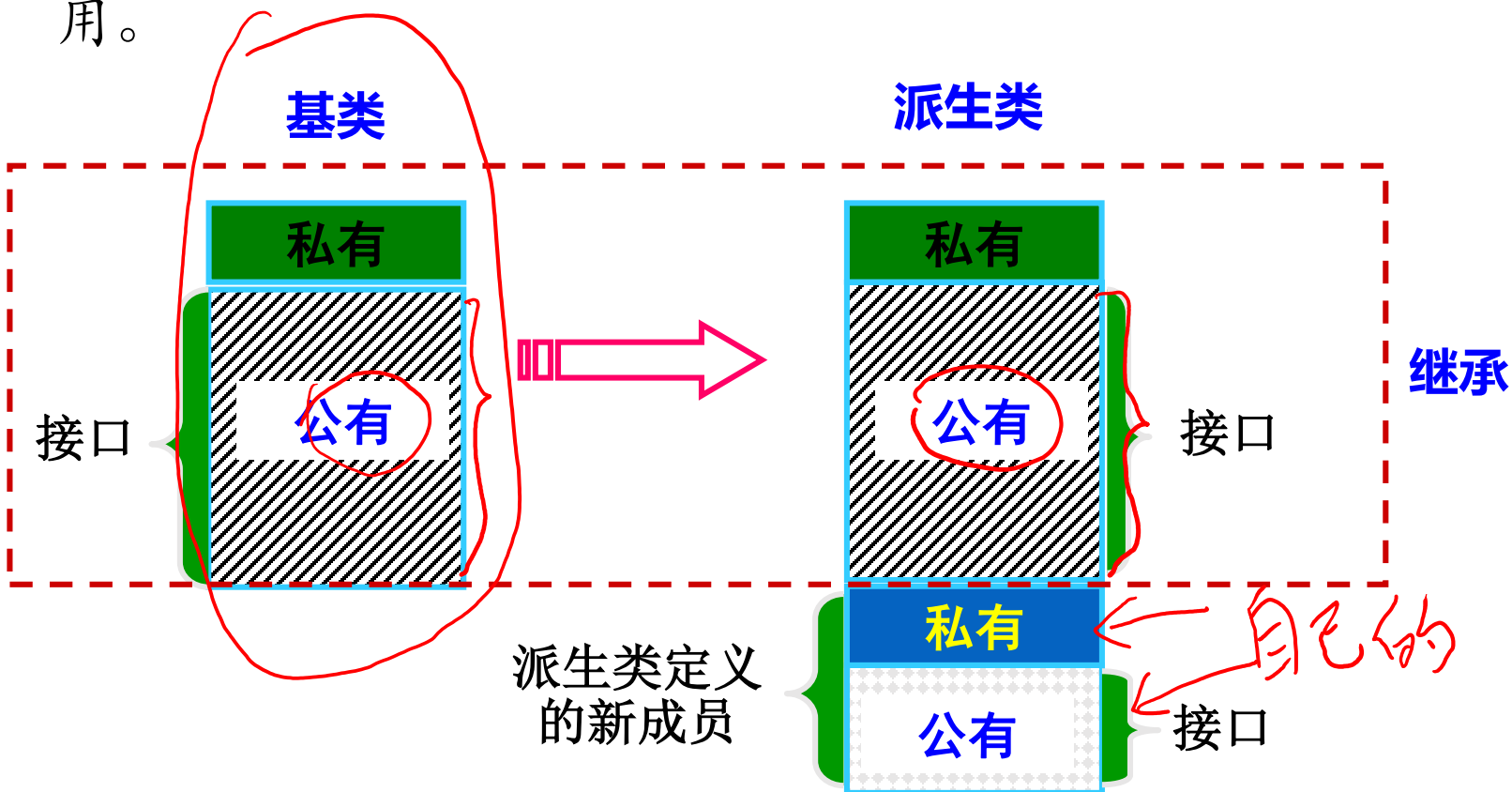
- A 派生类自动继承基类的数据成员、函数成员、赋值运算符; ~~X~~
- B 基类中没有指定访问说明符时，编译器将默认该说明符是public; ~~X~~
- C 派生类不会继承基类的构造函数，因此不能调用基类构造函数创建派生类对象的基类部分; ~~X~~
- D 派生类的构造函数可以调用特定的基类构造函数，间接访问基类的私有成员。 

提交

# 如何选择继承方式？

## ■public继承

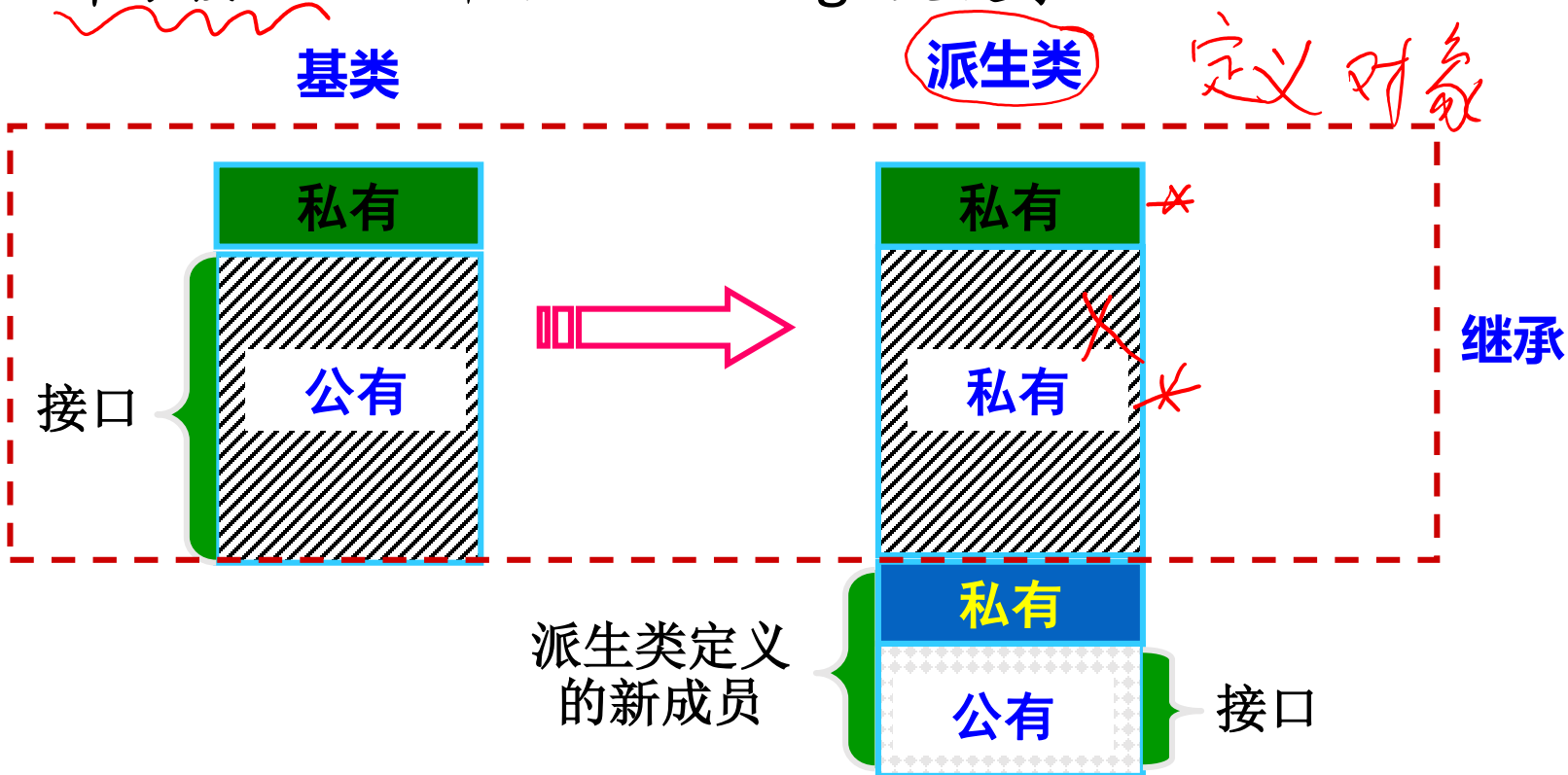
- 基类中公有成员仍能在派生类中保持公有。原接口可沿用。最常用。
- is-a: 基类对象能使用的地方，派生类对象也能使用。



# 如何选择继承方式?

## ■ private 继承

- is-implementing-in-terms-of(照此实现): 用基类接口实现派生类功能。移除了 is-a 关系。
- 通常不使用, 用组合替代。可用于隐藏/公开基类的部分接口。公开方法: using 关键字。



# 成员访问权限

- 基类中的私有成员，不允许在派生类成员函数中访问，也不允许派生类的对象访问它们。
  - 真正体现“基类私有”，对派生类也不开放其权限！
- 基类中的公有成员：
  - 允许在派生类成员函数中被访问
  - 若是使用`public`继承方式，则成为派生类公有成员，可以被派生类的对象访问；
  - 若是使用`private/protected`继承方式，则成为派生类私有/保护成员，不能被派生类的对象访问。若想让某成员能被派生类的对象访问，可在派生类`public`部分用关键字`using`声明它的名字。
- 基类中的保护成员
  - 与基类中的私有成员的不同在于：保护成员允许在派生类成员函数中被访问。

# 基类中的公有成员访问

```
#include <iostream>
using namespace std;

class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive1: public Base {}; // D1类的继承方式是public继承

int main() {
    Derive1 obj1;
    cout << "calling obj1.baseFunc()..." << endl;
    obj1.baseFunc(); // 基类接口成为派生类接口的一部分, 派生类对象可调用
    return 0;
}
```

# 基类中的公有成员访问

```
#include <iostream>
using namespace std;
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive2: private Base
{/// 私有继承, is-implementing-in-terms-of: 用基类接口实现派生类功能
public:
    void deriveFunc() {
        cout << "in Derive2::deriveFunc(),
                    calling Base::baseFunc()..." << endl;
        baseFunc(); /// 私有继承时, 基类接口在派生类成员函数中可以使用
    }
};

int main() {
    Derive2 obj2;
    cout << "calling obj2.deriveFunc()..." << endl;
    obj2.deriveFunc();
    ///obj2.baseFunc(); ERROR: 基类接口不允许派生类对象调用

    return 0;
}
```



# 基类中的公有成员访问

```
#include <iostream>
using namespace std;
class Base {
public:
    void baseFunc() { cout << "in Base::baseFunc()..." << endl; }
};

class Derive3: private Base { // B的私有继承
public:
    /// 私有继承时, 在派生类public部分声明基类成员名字
    using Base::baseFunc;
};

int main() {
    Derive3 obj3;
    cout << "calling obj3.baseFunc()..." << endl;
    obj3.baseFunc(); //基类接口在派生类public部分声明, 则派生类对象可调用
    return 0;
}
```

# 基类中的 私有，保护成员访问

```
#include <iostream>
using namespace std;
```

```
class Base{
private: *
    int a{0};
protected:
    int b{0}; △
};
```

```
class Derive : private Base{
public:
    void getA(){cout<<a<<endl;} ///编译错误，不可访问基类中私有成员
    void getB(){cout<<b<<endl;} ///可以访问基类中保护成员
};
```

```
int main()
{
    Derive d;
    d.getB();
    //cout<<d.b; ///编译错误，派生类对象不可访问基类中保护成员
    return 0;
}
```

# 基类中的 私有，公有成员访问

```
#include <iostream>
using namespace std;
```

```
class Base {
private:
    int data{0};
public:
    int getData(){ return data;}
    void setData(int i){ data=i;}
};
```

```
class Derive1 : private Base {
public:
    using Base::getData;
};
```

```
int main() {
    Derive1 d1;
    cout<<d1.getData();
    //d1.setData(10);
    //Base& b = d1;
    //b.setData(10);
}
```

///隐藏了基类的setData函数，不可访问

///不允许私有继承的向上转换

///否则可以绕过D1，调用基类的setData函数

b.setData(10);

# 基类成员访问权限与三种继承方式

## ■public继承

- 基类的公有成员，保护成员，私有成员作为派生类的成员时，都**保持**原有的状态。

## ■private继承

- 基类的公有成员，保护成员，私有成员作为派生类的成员时，都作为**私有**成员。

## ■protected继承

- 基类的公有成员，保护成员作为派生类的成员时，都成为**保护**成员，基类的私有成员仍然是**私有**的。

# 成员访问权限

基类私有

继承表		继承方法					
		public		private		protected	
基类中 成员类型	* public	OK	pub/yes	OK	prv/no	OK	pro/no
	* private	NO	prv/no	NO	prv/no	NO	prv/no
	* protected	OK	pro/no	OK	prv/no	OK	pro/no

派生类成员函数  
能否访问基类成员

Derive::func(...) { ? }

基类成员在派生类中的成员类型，  
派生类对象能否访问基类成员

Derive d; d.(?)

prv: private  
pro: protected  
pub: public

类似集合交运算(成员类型与继承类型之间取交)  
Order: public  $\supset$  protected  $\supset$  private

(不定项选择题) 为避免编译错误，下述代码中(1)处可以填写

```
#include <iostream>
using namespace std;
class A {
public:
    int a=1;
protected:
    int b=2;
private:
    int c=3;
};
```

```
class B: public A {
public:
    void print() {
        cout << b << endl;
    }
};
int main() {
    B obj_b;
    obj_b.print();
    cout << (1) << endl;
    return 0;
}
```

**A**

obj\_b.a ✓

B

obj\_b.b ✗

C

obj\_b.c ✗

提交

# 组合与继承

## ■ 组合与继承的优点：支持增量开发。

- 允许引入新代码而不影响已有代码正确性。

## ■ 相似：

- 实现代码重用。
- 将子对象引入新类(继承?)。
- 使用构造函数的初始化成员列表初始化。

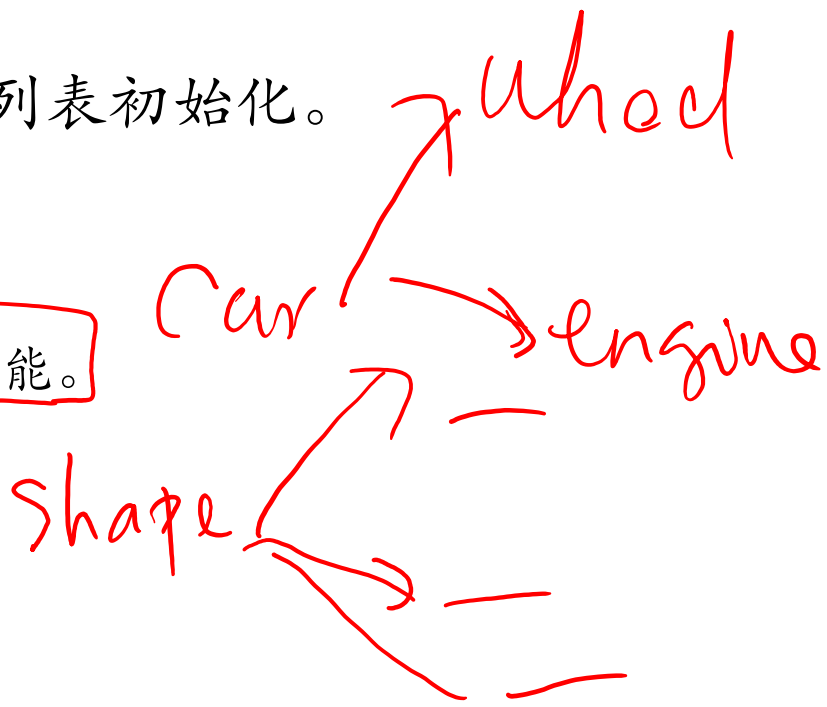
## ■ 不同：

### • 组合：

- 嵌入一个对象以实现新类的功能。
- has-a 关系。

### • 继承：

- 沿用已存在的类提供的接口。
- public 继承：is-a。
- private 继承：is-implementing-in-terms-of。



# 组合示例 has-a

```
#include <iostream>
using namespace std;
```

```
class Wheel{
public:
    void inflate(){
        cout<<"Wheel::inflate"<<endl;
    }
};
```

```
class Engine{
public:
    void start(){
        cout<<"Engine::start"<<endl;
    }
    void stop(){}
};
```

```
class Car{
public:
    Engine engine; ✓
    Wheel wheel[4]; ✓
};
```

```
int main()
{
    Car car;
    car.wheel[0].inflate();
    car.engine.start();
    return 0;
}
```

运行结果

Wheel::inflate  
Engine::start



# 继承示例 is-a

```
#include <iostream>
using namespace std;

class Pet{
public:
    void eat(){cout<<"Pet eat"<<endl;}
    void sleep(){}
};

class Duck : public Pet{
public:
    void eat(){cout<<"Duck eat"<<endl;}
};

int main()
{
    Duck duck;
    duck.eat(); ✖
    return 0;
}
```

运行结果

Duck eat

# 重写隐藏与重载

## ■ 重载(overload):

- 目的: 提供同名函数的不同实现, 属于静态多态。
- 函数名必须相同, 函数参数必须不同, 作用域相同  
(如位于同一个类中; 或同名全局函数)。

## ■ 重写隐藏(redefining):

- 目的: 在派生类中重新定义基类函数, 实现派生类的特殊功能。
- 屏蔽了基类的所有其它同名函数。
- 函数名必须相同, 函数参数可以不同

# 重写隐藏

- 重写隐藏发生时，基类中该成员函数的其他重载函数都将被屏蔽掉，不能提供给派生类对象使用
- 可以在派生类中通过 `using 类名::成员函数名;` 在派生类中“恢复”指定的基类成员函数（即去掉屏蔽），使之重新可用

# 函数重写隐藏示例

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
    void f() { cout << "B::f()\n"; }
    void f(int i) { cout << "Base::f(" << i << ")\n"; } // 重载
    void f(double d) { cout << "Base::f(" << d << ")\n"; } // 重载
    void f(T) { cout << "Base::f(T)\n"; } // 重载
};
class Derive : public Base {
public:
    void f(int i) { cout << "Derive::f(" << i << ")\n"; } // 重写隐藏
};
int main() {
    Derive d;
    d.f(10);
    * d.f(4.9); // 编译警告。执行自动类型转换。4.9 → 4
    // d.f(); // 被屏蔽, 编译错误
    // d.f(T()); // 被屏蔽, 编译错误
    return 0;
}
```

运行结果

D1::f(10)

D1::f(4)

4.9 → 4

# 恢复基类成员函数示例

```
#include <iostream>
using namespace std;
class T {};
class Base {
public:
    void f() { cout << "Base::f()\n"; }
    void f(int i) { cout << "Base::f(" << i << ")\n"; }
    void f(double d) { cout << "Base::f(" << d << ")\n"; }
    void f(T) { cout << "Base::f(T)\n"; }
};
```

```
class Derive : public Base {
public:
```

~~using~~ Base::f;

← 使用using 基类名::函数名;恢复基类函数

```
void f(int i) { cout << "Derive::f(" << i << ")\n"; }
```

```
};
```

```
int main() {
```

```
    Derive d;
```

```
    d.f(10);
```

✓ d.f(4.9); →

f(4)

✓ d.f();

✓ d.f(T());

```
    return 0;
```

```
}
```

## 运行结果

D1::f(10)

B::f(4.9)

B::f()

B::f(T)

# using关键字

## ■ using关键字除了可用于：

- 继承基类构造函数
- 恢复被屏蔽的基类成员函数

## ■ 还可用于：

- 指示命名空间，如：

using namespace std;

- 将另一个命名空间的成员引入当前命名空间，如：

using std::cout;

cout << endl;

- 定义类型别名，如：

using a = int;



进一步阅读： <https://en.cppreference.com/w/cpp/keyword/using>

# 多选题 1分

关于下列代码的说法正确的是 (\n为换行符)

```
#include <iostream>
using namespace std;
class A {
public:
    int data;
    A(int d)
    {cout << "A::A(" << d << ")\n";}
    void f(double d)
    {cout << "A::f(" << d << ")\n";}
protected:
    void g(){}
};
```

```
class B: public A {
public:
    int data{2017};
    using A::A;
    void f(){}
    void print(){
        cout << "data = " << data << endl;
    }
};

int main() {
    B b(6);
    b.print();
    return 0;
}
```

A main函数中可通过**b.g();**语句调用函数A::g()

B 去掉B类定义中的using A::A;语句, 程序会出现编译错误

C 程序的运行结果为A::A(6)\ndata=2017\n

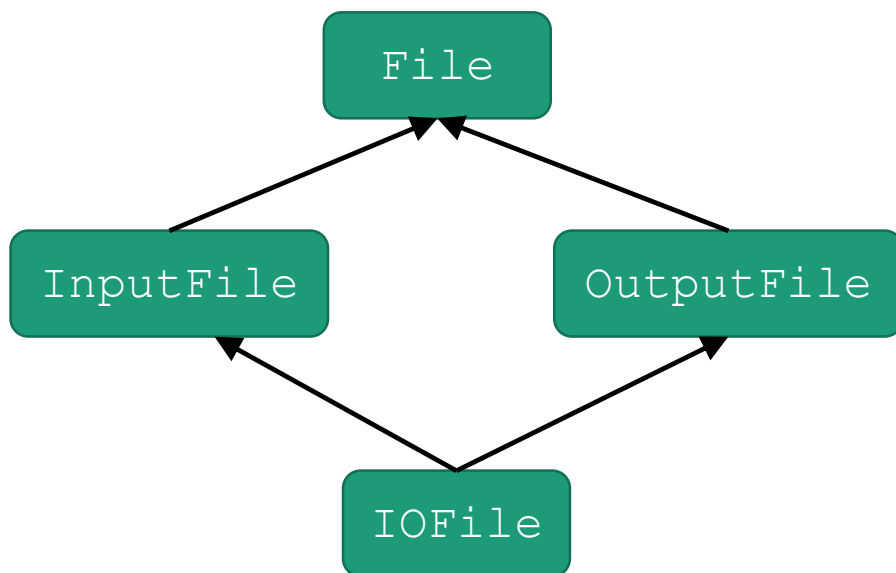
D main函数中可通过**b.f(17.315);**调用函数A::f(double d)

提交

# 多重继承

- 派生类同时继承多个基类
- 应用场景

```
class File{};  
class InputFile: public File{};  
class OutputFile: public File{};  
class IOFile: public InputFile, public OutputFile{};
```





# 多重继承问题

## ■ 数据存储

- 如果派生类D继承的两个基类A,B，是同一基类Base的不同继承，则A,B中继承自Base的数据成员会在D有两份独立的副本，可能带来数据冗余。

## ■ 二义性

- 如果派生类D继承的两个基类A,B，有同名成员a，则访问D中a时，编译器无法判断要访问的哪一个基类成员。

# 多重继承示例

```
#include <iostream>
using namespace std;
```

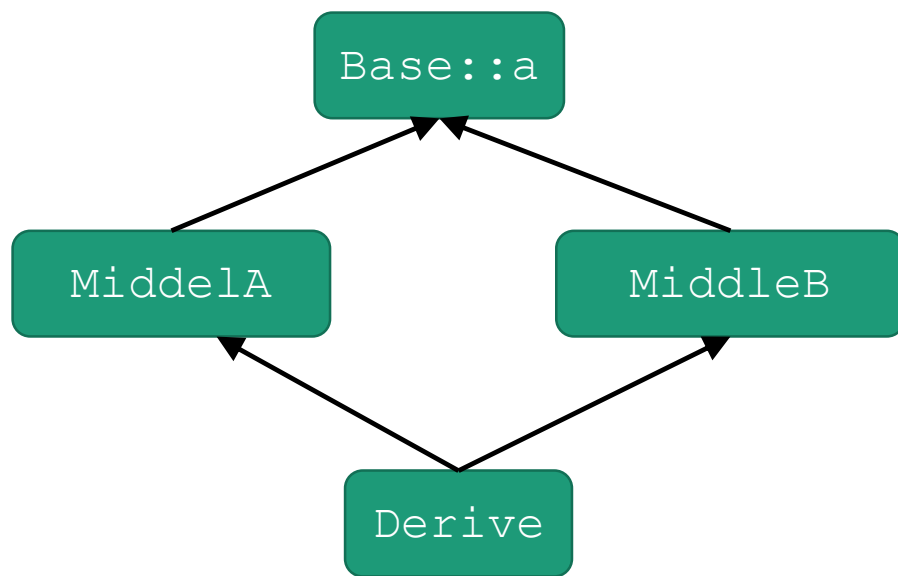
```
class Base {
public:
    int a{0};
};
```

```
class MiddleA : public Base {
public:
    void addA() { cout << "a=" << ++a << endl; };
    void bar() { cout << "A::bar" << endl; };
};
```

```
class MiddleB : public Base {
public:
    void addB() { cout << "a=" << ++a << endl; };
    void bar() { cout << "B::bar" << endl; };
};
```

```
class Derive : public MiddleA, public MiddleB{
};
```

# 多重继承



# 多重继承示例

```
int main() {
    Derive d;
    d.addA();          ///  
d.addB();          ///  
d.addB();          ///  
//cout << d.a;
    ///  
cout << d.MiddleA::a << endl;
    ///  
//d.bar();
    ///  
cout << d.MiddleB::a << endl;
    ///  
return 0;
}
```

# 课后阅读

## ■ 《C++编程思想》

- 继承与组合，p336-p361

## ■ 修改p7、p8代码，使得Wheel、Engine的构造函数带参数，实现各种构造函数版本

## ■ 编写小程序，探索public, private, protected继承对基类各种类型变量的访问权限

**结 束**